REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
01/09/2011	Final report	6/15/2010-6/14/2011

4. TITLE AND SUBTITLE

Helix Project Testbed: Towards the Self-Regenerative Incorruptible Enterprise

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA9550-10-1-0316

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Hao Chen

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of California, Davis
One Shields Avenue
Davis, California 95616

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research
875 North Randolph St.
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)

AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-OSR-VA-TR-2012-0789

12. DISTRIBUTION / AVAILABILITY STATEMENT

Public Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

As part of an AFOSR-sponsored MURI Initiative (BAA 06-028, Topic: Self-Regenerative Incorruptible Enterprise, Program Manager: Dr. Herklotz, 2007-2012), our team is realizing and refining the Helix vision of a self-regenerative architecture to protect enterprise-wide networked information systems. The funds were used to purchase and set up a Helix distributed testbed (HDT) consisting of servers, desktops, laptops, and mobile devices. The two primary and complementary goals of the HDT are: (1) to provide an enabling infrastructure to directly support our research and educational activities, and (2), to provide a realistic live testbed on which to prototype and evaluate the novel capabilities developed by the various research thrusts. We have purchased three rack-mounted multicore systems. Together with similar systems hosted at the other three teams on the same MURI project, they were aggregated to form the computational and storage backbone of the HDT —the data center or cloud concept—and provide services to the enterprise. Using the testbed, we have evaluated the security and performance of Noncespaces and DBTaint, techniques for protecting web services from cross-site scripting attacks.

15. SUBJECT TERMS

Testbed, Security, servers.

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Final Report

Grant/Contract Title: Helix Project Testbed: Towards the Self-Regenerative Incorruptible Enterprise
Grant/Contract Number: FA9550-10-1-0316
Principal Investigator: Hao Chen
Date: June 15, 2010 --- June 14, 2011

1 Introduction

As part of an AFOSR-sponsored MURI Initiative (BAA 06-028, Topic: Self-Regenerative Incorruptible Enterprise, Program Manager: Dr. Herklotz, 2007-2012), our team is realizing and refining the Helix vision of a self-regenerative architecture to protect enterprise-wide networked information systems.

2 Helix testbed

The funds were used to purchase and set up a Helix distributed testbed (HDT) consisting of servers, desktops, laptops, and mobile devices.

The two primary and complementary goals of the HDT are: (1) to provide an enabling infrastructure to directly support our research and educational activities, and (2), to provide a realistic live testbed on which to prototype and evaluate the novel capabilities developed by the various research thrusts.

We have purchased three rack-mounted multicore systems. Together with similar systems hosted at the other three teams on the same MURI project, they were aggregated to form the computational and storage backbone of the HDT—the data center or cloud concept—and provide services to the enterprise. In addition, we have purchased several lightweight mobile Internet devices (tablets, cell phones, smartphones, consumer-grade Internet devices) and projection devices to model the various access and presentation modalities by which enterprise-wide information will be accessed and manipulated. We anticipate that web-based technology, e.g., mashups, browsers and other clients, web and application servers, will become prevalent throughout the enterprise.

We have set up the testbed so that researchers that participate in Helix-related research activities have access to the resources across four sites, and have the ability to reserve a large fraction of the testbed for controlled at-scale experiments.

In summary, we have set up a flexible testbed in which the following representative types of activities and experiments can be carried out:

- Experiments that take advantage of the nature of the resources requested, e.g. understanding the performance implications of various multicore configurations, using multicore systems for security and/or program analysis.
- Experiments that require vast computational power, e.g. parameter space studies, large scale genetic algorithms for program repair.
- Experimental setup that mimics the operating conditions of large-scale enterprises, e.g., with critical services such as mail, file, database, and web servers.
- Honeypots and other monitoring services to gain an in-depth understanding of services under normal and anomalous conditions.

- Experimentation with mobile internet devices. This class of devices provides access point to various enterprise services. However, their relative small form factor and limited computational power present both challenges and opportunities for security research.

- Prototyping and evaluation of the Helix architecture at scale, including setting up fine-grained sensors and actuators across the enterprise, and/or performing live experiments in which servers are continuously monitored and repaired automatically.

3 Research accomplishments

3.1 Using randomization to defend against cross-site scripting attacks

As the Web evolves into a general purpose computing platform and becomes prevalent in consumer and DoD enterprise applications, attacks on Web applications will be even more widespread than today. Compared to traditional desktop applications, Web applications are much more difficult to secure because they involve both the server and the client. The server creates Web content that usually contains untrusted user input, and the client renders the content and executes the code in the content. Without protection, malicious user content could disclose confidential information and modify critical information of other victim users. In the context of a DoD enterprise system this could easily jeopardize the integrity of the overall mission.

Cross-site scripting (XSS) attack is a typical example of a web attack. To defend against these attacks, untrusted user content must be sanitized. Currently, most approaches try to sanitize untrusted user content at the server side. However, different browsers may parse the same Web content differently, partly in an attempt to tolerate or auto-correct malformed Web pages. Therefore, it is difficult, if not impossible, for the server to sanitize untrusted Web content so that it would be safe for any Web browsers. We need to rely on the client to enforce a security policy on the content (e.g., to deny or sanitize malicious content).

Our approach, Noncespaces [3, 2], lets the Web server and client collaborate to secure Web content. The Web server tracks the flow of untrusted user input and annotates such input in the generated Web content. The client, upon receiving the annotated Web content, enforces an appropriate security policy on the content. This approach, however, raises two challenges: First, how does the server track untrusted user content? Second, how does the server annotate untrusted user content in the Web page securely (i.e., how to prevent untrusted content from embedding fake annotations)?

The Helix Distributed Testbed was used for developing randomization in Web applications and information flow tracking in databases, and for evaluating their performance on a realistic environment. The mobile Internet devices was used to develop Web page sanitization and other policy enforcement in Web browsers and to evaluate their performance. Since mobile devices are becoming increasingly popular and more Web transactions are moving to mobile devices, it is imperative to protect mobile Web browsers from untrusted-data attacks.

3.2 Information flow tracking in databases

Previous research has addressed how to track information flow within a single application, e.g., by assigning a taint bit to every byte in the application. This approach, however, is insufficient for our purpose. This is because most Web services consist of multiple applications. For example, a typical Web service consists of a Web application for creating Web content and a database application for storing data. Currently, no major database supports information flow tracking, so the Web application has to assume that all data

coming from the database are either all trusted or all untrusted. In other words, as soon as a data enters the database, its trustworthiness is lost for ever. As a result, this causes high false negative or false positive in identifying untrusted data.

One might try to solve this problem by using system-wide information flow tracking. However, that would be overkill, because we only need to track information flow within a few applications. System-wide information flow tracking has severe performance penalty, partly because it fails to take advantage of application semantics. By contrast, since we know the semantics of database operations, we could implement information flow much more efficiently.

We have developed information flow tracking in databases, called DBTaint [1]. DBTaint three parts:

- Expand the database such that each piece of data is tagged with a trust bit. The trust bits propagate through database operations.
- Design an interface for the database client to set and get the trust bits of data.
- Design a database client library that sets the gets the trust bits of data automatically. This allows the programmer to get the benefit of information flow tracking across databases without extra work (except for substituting our library for the default database interface). Using randomization to annotate untrusted content and to enforce security policies

4 Evaluation

To evaluate the effectiveness and overhead of Noncespaces we conducted several experiments. We evaluated the security of Noncespaces to ensure that it is able to prevent a wide variety of XSS attacks. Our performance evaluation measures the costs of Noncespaces from both the client's and server's points of view.

4.1 Security

4.1.1 TikiWiki Case Study

We tested Noncespaces against six XSS exploits targeting two vulnerable applications. The exploits were crafted to exhibit the various forms that an XSS attack may take [3]. The applications used in this evaluation were a version of TikiWiki with a number of XSS vulnerabilities and Trustify, a custom web application that we developed to cover all the major XSS vectors.

We began by developing policies for each application. Because TikiWiki was developed before Noncespaces existed, it illustrates the applicability of Noncespaces to existing applications. We implemented a straightforward 37-rule, static-dynamic policy that allows unconstrained static content but restricts the capabilities of dynamic content to that of BBCode. We also had to add exceptions for trusted content that TikiWiki generates dynamically by design, such as names and values of form elements, certain JavaScript links implementing collapsible menus, and custom style sheets based on user preferences.

For Trustify, our custom web application, we implemented a policy that does not take advantage of the static-dynamic model. Instead, the policy takes advantage of Noncespaces's ability to thwart node splitting attacks to implement an ancestry-based sandbox policy similar to the noexecute policy described in BEEP.

For each of the exploits we first verified that each exploit succeeded without Noncespaces enabled. We then enabled Noncespaces and verified that all exploits were blocked as policy violations.

4.1.2 LifeType Case Study

To gain more insight into the work involved in porting existing applications to Noncespaces, we ported LifeType, a popular blog application, to work with Noncespaces. LifeType is a mature, full-featured blog application consisting of 155K lines of PHP and XHTML code. Enabling Noncespaces required changes to only 180 lines of code. The majority of code changes occurred in LifeType's HTTP header handling. These changes were necessary because Noncespaces needs to include its own headers before any content is sent to the client.

We developed a static-dynamic policy for LifeType that attempts to restrict untrusted content to a minimal set of capabilities. Using our proxy's training mode, it took approximately 4 hours to exercise a significant portion of LifeType's functionality and to manually refine generated rules that were overly general. We then went through our functionality exercise again to ensure that we did not prohibit any legitimate behavior.

To test the effectiveness of our LifeType policy, we introduced XSS vulnerabilities into the application. We used the XSS Cheat Sheet to craft 100 XSS exploits. We then tested each exploit in Opera 9.27. Before applying Noncespaces, 50 of the exploits were successful. The remaining 50 exploits were unsuccessful against Opera because they exploit functionality unique to some other browser (such as executing JavaScript by invoking the `mocha:` protocol scheme present in older Netscape versions). After we applied Noncespaces, Noncespaces blocked 98 of the 100 exploits as either policy violations or XHTML parsing errors. These results give us confidence in our policy's ability to recognize exploits while allowing intended behavior and in Noncespaces's ability to block exploits that target multiple browsers. Since Noncespaces processes exploitable web pages before the browser renders them, many exploits that would have been incompatible with the browser were blocked by Noncespaces before they reached the browser. Neither of the two exploits that were not blocked resulted in a successful XSS attack: one was rendered as text, the other as a comment. That neither exploit caused a policy violation does not indicate a limitation of our approach. Our browser-agnostic prototype proxy implementation targets XHTML compliant browsers, as discussed previously. Neither exploit was valid XHTML.

4.2 Performance

Our performance evaluation seeks to measure the overhead of Noncespaces in terms of response latency and server throughput. Our test infrastructure consisted of the applications that we used for our security evaluation running in a VMware virtual machine with 512 MB RAM running Fedora Core 3, Apache 2.0.52, and mod_php 5.2.6. The virtual machine ran on an Intel Pentium 4 3.2 GHz machine with 1 GB RAM running Ubuntu 7.10. Our client machine was an Intel Pentium 4 2 GHz machine with 256 MB RAM running Ubuntu 8.10 Server. These results represent an upper bound on performance penalty as we have spent no effort optimizing our Noncespaces prototype. In each test we used `ab` to retrieve an application page 1000 times. We varied the number of concurrent requests between 1, 5, 10, and 15, and the configuration of the client and server between the following:

- Baseline: measures original web application performance before applying Noncespaces.
- Randomization Only: measures impact of Noncespaces randomization on server without policy validation on client-side.
- Full Enforcement: measures the end-to-end impact of Noncespaces.

We ran three trials with each test configuration against both the TikiWiki and Lifetype applications. We report the mean, median, and standard deviation of results over all trials. The server virtual machine was

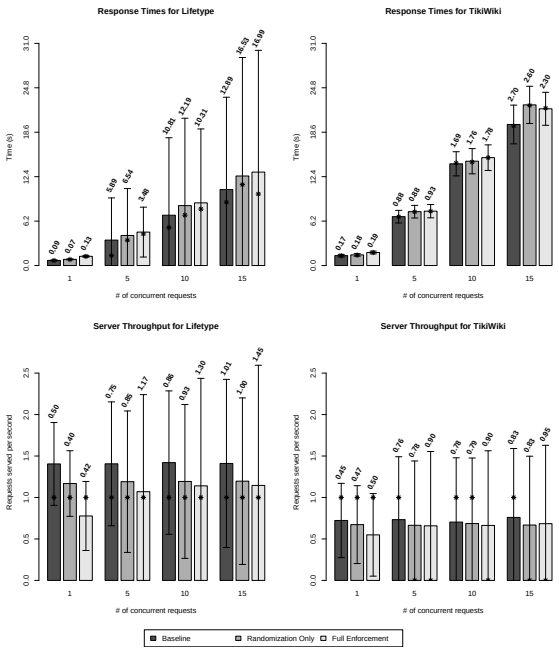

Figure 1: Performance of Noncespaces

rebooted between tests. The target page was prefetched once before the test to warm up the systems' caches to prevent any one-time costs (such as compiling the NSmarty templates) from skewing our results. Our results are shown in Figure 1.

The graphs of response latency show that enabling Noncespaces randomization on the server increased response time by (at most) 14% for TikiWiki and 20% for LifeType. Enabling the policy checking proxy resulted in response times that were (at most) 32% higher than the baseline response time for TikiWiki and 80% higher for LifeType. Though the overhead may appear significant at first glance, during interactive use latency typically increased by no more than 0.6 seconds.

We also examine the effect of Noncespaces on server throughput. With randomization enabled throughput is reduced by about 10% for TikiWiki and 20% for LifeType. After enabling policy checking, the throughput of both TikiWiki and LifeType decreases by an additional 3% for higher numbers of concurrent requests. Because policy checking is performed on the client side the effect of policy checking on server throughput is minimized when multiple clients make requests simultaneously.

Publications

[1] Benjamin Davis and Hao Chen. ``DBTaint: Cross-Application Information Flow Tracking via Data bases". In: *USENIX Conference on Web Application*. Boston, MA, 0623-24, 2010.

[2] Matthew Van Gundy and Hao Chen. ``Noncespaces: Using Randomization to Defeat Cross-Site Scripting Attacks". In: *Computer and Security* (2012).

[3] Matthew Van Gundy and Hao Chen. ``Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks". In: *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Feb. 2009, pp. 55–67.